Self-Verifying Concurrent Programming

Peter Welch\textsuperscript{a}, Matt Pedersen\textsuperscript{b}, Fred Barnes\textsuperscript{a} and Carl Ritson\textsuperscript{a}

\textsuperscript{a} School of Computing, University of Kent, UK
\textsuperscript{b} School of Computer Science, UNLV, USA

IFIP WG2.4, Cape May, 29\textsuperscript{th}. September, 2011
occam-$\pi$, the process algebra

**Aim:**

To enable formal verification of *occam-$\pi$* programs to be conducted within the language itself ... *as a matter of course by the programmer.*

**How:**

Extend *occam-$\pi$* with *verification qualifiers* and *assertions*. Modify the compiler to generate *(minimal)* $CSP_M$ code from programs using these qualifiers and assertions, bounce this off the *FDR* model checker and report back in terms of the source code.
occam-π, the process algebra

Why?

Because, now, we can ... it’s needed ... and it’s time!
Example: autonomous robot component

The following example has been developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.
Example: autonomous robot component

The following example has been developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.

**Device**: real-time controller for 8 channels (4 input, 4 output).
**Example: autonomous robot component**

Device: real-time controller for 8 channels (4 input, 4 output).

There are 3 sub-components: **P0** *(weapons systems)*, **P1** *(vision processing)* and **P2** *(motion stabiliser)*.

They exchange information over internal channels *(ask, ans, ping)* and coordinate actions with an internal barrier *(bar)*.
Example: autonomous robot component

- **CSP** semantics apply. **Channel communication** is unbuffered (sender waits for receiver and vice-versa). Any process reaching a barrier waits for all processes to reach the barrier.

They exchange information over internal channels (ask, ans, ping) and coordinate actions with an internal barrier (bar).
Behaviour: two representations

Device

P0

\[\text{ask} \quad \text{ans} \quad \text{bar}\]

P1

P2

\[\text{ping}\]

\[a_0 \quad b_0 \quad c_0 \quad a_1 \quad b_1 \quad c_1 \quad d_0 \quad d_1\]

\textbf{occam-\pi}: for compiling to a runnable system.

\textbf{CSP}: for formal analysis.

\[\text{occam-\pi}: \text{for compiling to a runnable system.} \]

\[\text{CSP}: \text{for formal analysis.} \]

\[\text{memory overheads} \leq 32 \text{ bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.} \]
Behaviour: *one representation*

\[
\begin{array}{c}
\text{a0} \quad \text{b0} \quad \text{c0} \\
\downarrow \quad \downarrow \quad \downarrow \\
\text{a1} \quad \text{b1} \quad \text{c1} \\
\end{array}
\]

**Device**

\[
\begin{array}{c}
\text{P0} \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\text{ask} \quad \text{ans} \quad \text{ping} \\
\end{array}
\]

### occam-$\pi$:
- for compiling to a runnable system.
  - memory overheads $\leq 32$ bytes per process / synchronisation overheads of order tens of nanoseconds / eats multicore nodes for breakfast.

### occam-$\pi$:
- for formal analysis.
  - verify qualifiers and (FDR) assertions + other (simple) formal reasoning.
**Behaviour: what are we looking for?**

**deadlock:** *might* it ever stop?  
[e.g. $P_0$ and $P_2$ want to synchronise on $bar$, but $P_1$ wants to $ping$.]

**livelock:** *might* it get busy ... but refuse all external signals?  
[e.g. $P_0$, $P_1$ and $P_2$ start engaging in an infinite sequence of internal channel or barrier synchronisations (on $ask$, $ans$, $ping$ and $bar$).]
**Behaviour: what are we looking for?**

**safety:** *might* it ever engage in an incorrect sequence of external signals?

**liveness:** *will* it engage in correct sequences of external signals, as required?
For the behaviour analysis in this example, data values and computations are not relevant. For simplicity, they are omitted in these codes, with all message content abstracted to zero.
**Behaviour: **\texttt{occam-\pi} (executable)

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer (will depend on x and y)
      b0 ? z
      SYNC bar    -- wait for the others
      c0 ! 0
```

```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
        BARRIER bar)

    WHILE TRUE
        INT x, y, z:
        SEQ
            ask ! 0     -- ask question
            ans ? x     -- wait for answer
            a1 ? y
            b1 ? z
            SYNC bar    -- wait for the others
            c1 ! 0
            ping ! 0    -- update neighbour
            ;
PROC P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar -- wait for the others
      d0 ! 0
      ping ? x -- receive update
      SYNC bar -- wait for the others
      d1 ! 0
      ping ? x -- receive update

::
PROC Device (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
CHAN INT ask, ans, ping:
BARRIER bar:
PAR ENROLL bar
P0 (a0?, b0?, c0!, ask?, ans!, bar)
P1 (a1?, b1?, c1!, ask!, ans?, ping!, bar)
P2 (d0!, d1!, ping?, bar)
Informal Intuitive

**Behaviour:** *occam-π (executable)*

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)

```occam-π
WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x     -- take question
    a0 ? y
    ans ! 0     -- return answer
    b0 ? z
    SYNC bar    -- wait for others
    c0 ! 0
```

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)

```occam-π
WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0     -- ask question
    ans ? x     -- wait for answer
    a1 ? y
    b1 ? z
    SYNC bar    -- wait for the others
    c1 ! 0
    ping ! 0    -- update neighbour
```

**PROC P2** (CHAN INT d0!, d1!, ping?, BARRIER bar)

```occam-π
WHILE TRUE
  INT x:
  SEQ
    SYNC bar    -- wait for others
    d0 ! 0
    ping ? x    -- receive update
    SYNC bar    -- wait for others
    d1 ! 0
    ping ? x    -- receive update
```

**What patterns of external (blue) signalling are possible from Device?**
Informal

Intuitive

Behaviour: \texttt{occam-\pi} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x  -- take question
      a0 ? y   -- return answer
      ans ! 0  -- wait for the others
      b0 ? z
      SYNC bar
      c0 ! 0
: 

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0  -- ask question
      ans ? x  -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar  -- wait for the others
      c1 ! 0
      ping ! 0  -- update neighbour
: 

PROC P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar  -- wait for others
      d0 ! 0
      ping ? x  -- receive update
      SYNC bar  -- wait for others
      d1 ! 0
      ping ? x  -- receive update
: 
\end{verbatim}

What's first?
Informal
Intuitive

Behaviour: \texttt{occam-\pi} (executable)

**PROC P0** (CHAN INT \texttt{a0?}, \texttt{b0?}, \texttt{c0!}, \texttt{ask?}, \texttt{ans!}, BARRIER \texttt{bar})

\begin{verbatim}
WHILE TRUE
  INT x, y, z:
  SEQ
  ask ? x    \hfill -- take question
  a0 ? y
  ans ! 0    \hfill -- return answer
  b0 ? z
  SYNC bar   \hfill -- wait for others
  c0 ! 0
  :
\end{verbatim}

**PROC P1** (CHAN INT \texttt{a1?}, \texttt{b1?}, \texttt{c1!}, \texttt{ask!}, \texttt{ans?}, \texttt{ping!}, BARRIER \texttt{bar})

\begin{verbatim}
WHILE TRUE
  INT x, y, z:
  SEQ
  ask ! 0    \hfill -- ask question
  ans ? x    \hfill -- wait for answer
  a1 ? y
  b1 ? z
  SYNC bar   \hfill -- wait for the others
  c1 ! 0
  ping ! 0   \hfill -- update neighbour
  :
\end{verbatim}

**PROC P2** (CHAN INT \texttt{d0!}, \texttt{d1!}, \texttt{ping?}, BARRIER \texttt{bar})

\begin{verbatim}
WHILE TRUE
  INT x:
  SEQ
  SYN C bar    \hfill -- wait for others
  d0 ! 0
  ping ? x    \hfill -- receive update
  SYN C bar   \hfill -- wait for others
  d1 ! 0
  ping ? x    \hfill -- receive update
  :
\end{verbatim}

What's first?
Behaviour: \texttt{occam-\Pi} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
        BARRIER bar)
        WHILE TRUE
          INT x, y, z:
          SEQ
            ask ? x     -- take question
            a0 ? y      -- return answer
            ans ! 0     -- wait for the others
            b0 ? z
            SYNC bar    -- wait for others
            c0 ! 0

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
        BARRIER bar)
        WHILE TRUE
          INT x, y, z:
          SEQ
            ask ! 0     -- ask question
            ans ? x     -- wait for answer
            a1 ? y
            b1 ? z
            SYNC bar    -- wait for the others
            c1 ! 0
            ping ! 0    -- update neighbour

PROC P2 (CHAN INT d0!, d1!, ping?,
        BARRIER bar)
        WHILE TRUE
          INT x:
          SEQ
            SYNC bar    -- wait for others
            d0 ! 0
            ping ? x    -- receive update
            d1 ! 0
            ping ? x    -- receive update

What's first?

\[ \texttt{a0} \]
\[ \langle \texttt{a0} \rangle \]
\end{verbatim}
Behaviour: \textit{occam-\pi} (executable)

\begin{enumerate}
\item \textbf{PROC P0} \texttt{(CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)}
  \begin{small}
  \begin{align*}
  \text{WHILE TRUE} \\
  \text{INT x, y, z:} \\
  \text{SEQ} \\
  \quad \text{ask} ? x & \quad \text{-- take question} \\
  \quad a0 ? y & \quad \text{-- return answer} \\
  \quad \text{ans} ! 0 & \quad \text{-- return answer} \\
  \quad b0 ? z & \quad \text{-- return answer} \\
  \quad \text{SYNC bar} & \quad \text{-- wait for others} \\
  \quad c0 ! 0 & \quad \text{-- wait for others} \\
  \end{align*}
  \end{small}
\item \textbf{PROC P1} \texttt{(CHAN INT a1?, b1?, cl!, ask!, ans?, ping!, BARRIER bar)}
  \begin{small}
  \begin{align*}
  \text{WHILE TRUE} \\
  \text{INT x, y, z:} \\
  \text{SEQ} \\
  \quad \text{ask} ! 0 & \quad \text{-- ask question} \\
  \quad \text{ans} ? x & \quad \text{-- wait for answer} \\
  \quad a1 ? y & \quad \text{-- wait for answer} \\
  \quad b1 ? z & \quad \text{-- wait for answer} \\
  \quad \text{SYNC bar} & \quad \text{-- wait for the others} \\
  \quad cl ! 0 & \quad \text{-- update neighbour} \\
  \quad \text{ping} ! 0 & \quad \text{-- update neighbour}
  \end{align*}
  \end{small}
\item \textbf{PROC P2} \texttt{(CHAN INT d0!, d1!, ping?, BARRIER bar)}
  \begin{small}
  \begin{align*}
  \text{WHILE TRUE} \\
  \text{INT x:} \\
  \text{SEQ} \\
  \quad \text{SYNC bar} & \quad \text{-- wait for others} \\
  \quad d0 ! 0 & \quad \text{-- wait for others} \\
  \quad \text{ping} ? x & \quad \text{-- receive update} \\
  \quad \text{SYNC bar} & \quad \text{-- wait for others} \\
  \quad d1 ! 0 & \quad \text{-- receive update} \\
  \quad \text{ping} ? x & \quad \text{-- receive update}
  \end{align*}
  \end{small}
\end{enumerate}

What's second?
Informal Intuitive

Behaviour: **occam-π (executable)**

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer
      b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0
  ::

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar    -- wait for the others
      c1 ! 0
      ping ! 0    -- update neighbour
  ::

PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update
  ::
```

What's second? <a0>
**Informal Intuitive**

**Behaviour:** `occam-π (executable)`

**PROC P0** (CHAN INT `a0?, b0?, c0!, ask?, ans!`, BARRIER `bar`)

WHILE TRUE

INT `x, y, z`:

SEQ

ask ? `x` -- take question

`a0 ? y` -- return answer

`b0 ? z` -- wait for others

SYNC `bar`

`c0 ! 0` -- update neighbour

::

**PROC P1** (CHAN INT `a1?, b1!, c1!, ask!, ans?, ping!`, BARRIER `bar`)

WHILE TRUE

INT `x, y, z`:

SEQ

`ask ! 0` -- ask question

ansi ? `x` -- wait for answer

`a1 ? y` -- wait for the others

`bl ? z` -- update neighbour

SYNC `bar`

`cl ! 0`

ping ! `0` -- update neighbour

::

**PROC P2** (CHAN INT `d0!, d1!, ping?`, BARRIER `bar`)

WHILE TRUE

INT `x`:

SEQ

SYNC `bar` -- wait for others

`d0 ! 0`

ping ? `x` -- receive update

SYNC `bar` -- wait for others

`d1 ! 0`

ping ? `x` -- receive update

::

What's second?

- b0
- a1

<>
Informal
Intuitive

Behaviour: **occam-\(\pi\) (executable)**

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ? x  -- take question
a0 ? y
ans ! 0  -- return answer
b0 ? z
SYNC bar  -- wait for others
c0 ! 0

::

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

ask ! 0  -- ask question
ans ? x  -- wait for answer
a1 ? y
b1 ? z
SYNC bar  -- wait for the others
c1 ! 0
ping ! 0  -- update neighbour

::

**PROC P2** (CHAN INT d0!, d1!, ping?, BARRIER bar)

WHILE TRUE

INT x:

SEQ

SYNC bar  -- wait for others
d0 ! 0
ping ? x  -- receive update
SYNC bar  -- wait for others
d1 ! 0
ping ? x  -- receive update

::

If  b0  second, then?

\(<a0, b0>\)
Informal Intuitive

Behaviour: occam-π (executable)

PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
          BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x    -- take question
      a0 ? y
      ans ! 0    -- return answer
      b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0
  :

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
          BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0    -- ask question
      ans ? x    -- wait for answer
      a1 ? y
      bl ? z
      SYNC bar    -- wait for the others
      cl ! 0
      ping ! 0    -- update neighbour
  :

PROC P2 (CHAN INT d0!, d1!, ping?,
          BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update
  :

If $b_0$ second, then?

$\langle a_0, b_0, a_1 \rangle$
Informal

Intuitive

Behaviour: \textit{occam-π} (executable)

\textbf{PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)}

\textbf{WHILE TRUE}

\textbf{INT x, y, z:}

\textbf{SEQ}

\begin{itemize}
  \item \texttt{ask} ? \texttt{x} \quad \text{-- take question}
  \item \texttt{a0} ? \texttt{y} \quad \text{-- return answer}
  \item \texttt{b0} ? \texttt{z} \quad \text{-- wait for others}
  \item \texttt{SYNC bar} \quad \text{-- wait for others}
  \item \texttt{c0} ! \texttt{0} \quad \text{-- update neighbour}
\end{itemize}

\textbf{PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)}

\textbf{WHILE TRUE}

\textbf{INT x, y, z:}

\textbf{SEQ}

\begin{itemize}
  \item \texttt{ask} ! \texttt{0} \quad \text{-- ask question}
  \item \texttt{ans} ? \texttt{x} \quad \text{-- wait for answer}
  \item \texttt{a1} ? \texttt{y} \quad \text{-- ask question}
  \item \texttt{bl} ? \texttt{z} \quad \text{-- wait for the others}
  \item \texttt{SYNC bar} \quad \text{-- wait for the others}
  \item \texttt{c1} ! \texttt{0} \quad \text{-- update neighbour}
  \item \texttt{ping} ! \texttt{0} \quad \text{-- update neighbour}
\end{itemize}

\textbf{PROC P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)}

\textbf{WHILE TRUE}

\textbf{INT x:}

\textbf{SEQ}

\begin{itemize}
  \item \texttt{SYNC bar} \quad \text{-- wait for others}
  \item \texttt{d0} ! \texttt{0} \quad \text{-- wait for others}
  \item \texttt{ping} ? \texttt{x} \quad \text{-- receive update}
  \item \texttt{SYNC bar} \quad \text{-- wait for others}
  \item \texttt{d1} ! \texttt{0} \quad \text{-- receive update}
  \item \texttt{ping} ? \texttt{x} \quad \text{-- receive update}
\end{itemize}

\textbf{If} \textbf{\texttt{b0}} \textbf{second, then?}

\textbf{\texttt{al} then \texttt{bl}}

\textbf{<a0, b0, a1, b1>
Informal Intuitive

**Behaviour:** \texttt{occam-\pi} (executable)

**PROC P0** (CHAN INT \texttt{a0}, \texttt{b0}, \texttt{c0}, \texttt{ask}, \texttt{ans},
BARRIER \texttt{bar})

\textbf{WHILE TRUE}

\textbf{INT} x, y, z:

\textbf{SEQ}

\texttt{ask ? x} \quad \text{-- take question}
\texttt{a0 ? y} \quad \text{-- return answer}
\texttt{b0 ? z} \quad \text{-- wait for others}
\texttt{SYNC bar} \quad \text{-- wait for others}
\texttt{c0 ! 0}

**PROC P1** (CHAN INT \texttt{a1}, \texttt{b1}, \texttt{c1}, \texttt{ask}, \texttt{ans}, \texttt{ping},
BARRIER \texttt{bar})

\textbf{WHILE TRUE}

\textbf{INT} x, y, z:

\textbf{SEQ}

\texttt{ask ! 0} \quad \text{-- ask question}
\texttt{ans ? x} \quad \text{-- wait for answer}
\texttt{a1 ? y} \quad \texttt{bl ? z}
\texttt{SYNC bar} \quad \text{-- wait for the others}
\texttt{c1 ! 0} \quad \text{-- update neighbour}
\texttt{ping ! 0}

**PROC P2** (CHAN INT \texttt{d0}, \texttt{d1}, \texttt{ping},
BARRIER \texttt{bar})

\textbf{WHILE TRUE}

\textbf{INT} x:

\textbf{SEQ}

\texttt{SYNC bar} \quad \text{-- wait for others}
\texttt{d0 ! 0}
\texttt{ping ? x} \quad \text{-- receive update}
\texttt{SYNC bar} \quad \text{-- wait for others}
\texttt{d1 ! 0}
\texttt{ping ? x} \quad \text{-- receive update}

\textbf{If} \quad \texttt{b0} \quad \text{second, then?}
\begin{itemize}
  \item \texttt{al} \quad \text{then}
  \item \texttt{bl}
\end{itemize}
\textbf{<a0, b0, a1, b1>}

\begin{itemize}
  \item \texttt{b0}
  \item \texttt{al}
  \item \texttt{bl}
\end{itemize}
Informal Intuitive

Behaviour: \texttt{occam-\pi} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer
      b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0

::

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar    -- wait for the others
      c1 ! 0
      ping ! 0    -- update neighbour

::

PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update

::

\end{verbatim}

What's second? 

- \texttt{b0}
- \texttt{a1}

\texttt{<a0>}

backtracking ...
**Informal Intuitive**

**Behaviour:** \textit{occam-\pi} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
         BARRIER bar)
  WHILE TRUE
      INT x, y, z:
      SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer
      b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
      INT x, y, z:
      SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      bl ? z
      SYNC bar    -- wait for the others
      cl ! 0
      ping ! 0    -- update neighbour

PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
      INT x:
      SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update

If \textbf{a1} second, then?
\end{verbatim}

\textbf{<a0, a1>}
Informal Intuative

Behaviour: \texttt{occam-\pi} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
        BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x     -- take question
      a0 ? y
      ans ! 0     -- return answer
      b0 ? z
      SYNC bar    -- wait for others
      c0 ! 0

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
        BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0     -- ask question
      ans ? x     -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar    -- wait for the others
      c1 ! 0
      ping ! 0    -- update neighbour

PROC P2 (CHAN INT d0!, d1!, ping?,
        BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar    -- wait for others
      d0 ! 0
      ping ? x    -- receive update
      SYNC bar    -- wait for others
      d1 ! 0
      ping ? x    -- receive update

<\texttt{a0, a1}>
If \texttt{a1} second, then? \texttt{b0} and \texttt{b1} *
\end{verbatim}
Informal Intuitive

**Behaviour: occam-π (executable)**

**PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)

WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x     -- take question
    a0 ? y
    ans ! 0     -- return answer
    b0 ? z
    SYNC bar    -- wait for others
    c0 ! 0

**PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)

WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0     -- ask question
    ans ? x     -- wait for answer
    a1 ? y
    b1 ? z
    SYNC bar    -- wait for the others
    c1 ! 0
    ping ! 0    -- update neighbour

**PROC P2** (CHAN INT d0!, d1!, ping?, BARRIER bar)

WHILE TRUE
  INT x:
  SEQ
    SYNC bar    -- wait for others
    d0 ! 0
    ping ? x    -- receive update
    SYNC bar    -- wait for others
    d1 ! 0
    ping ? x    -- receive update

If \(a1\) second, then?

\(b0\) and \(b1\)

\(<a0, a1, b0, b1>\)

\(<a0, a1, b1, b0>\)
Informal
Intuitive

Behaviour: **occam-π (executable)**

---

**PROCP0** (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

- ask ? x -- take question
- a0 ? y
- ans ! 0 -- return answer
- b0 ? z

SYNC bar -- wait for others

- c0 ! 0

:

**PROCP1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)

WHILE TRUE

INT x, y, z:

SEQ

- ask ! 0 -- ask question
- ans ? x -- wait for answer
- a1 ? y
- b1 ? z

SYNC bar -- wait for the others

- cl ! 0

ping ! 0 -- update neighbour

:

**PROCP2** (CHAN INT d0!, d1!, ping?, BARRIER bar)

WHILE TRUE

INT x:

SEQ

- SYNC bar -- wait for others
- d0 ! 0
- ping ? x -- receive update

SYNC bar -- wait for others

- d1 ! 0
- ping ? x -- receive update

:

---

If \(a_1\) second, then?

\(<a_0, a_1, b_0, b_1>\)

\(<a_0, a_1, b_1, b_0>\)
Informal

Intuitive

**Behaviour: occam-π (executable)**

- **PROC P0** (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  - WHILE TRUE
  - INT x, y, z:
  - SEQ
    - ask ? x  -- take question
    - a0 ? y  -- return answer
    - ans ! 0
    - b0 ? z
    - \( \rightarrow \) SYNC bar  -- wait for others
    - c0 ! 0

- **PROC P1** (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)
  - WHILE TRUE
  - INT x, y, z:
  - SEQ
    - ask ! 0  -- ask question
    - ans ? x  -- wait for answer
    - a1 ? y
    - b1 ? z
    - \( \rightarrow \) SYNC bar  -- wait for the others
    - c1 ! 0
    - ping ! 0  -- update neighbour

- **PROC P2** (CHAN INT d0!, d1!, ping?, BARRIER bar)
  - WHILE TRUE
  - INT x:
  - SEQ
    - \( \rightarrow \) SYNC bar  -- wait for others
    - d0 ! 0
    - ping ? x  -- receive update
    - \( \rightarrow \) SYNC bar  -- wait for others
    - d1 ! 0
    - ping ? x  -- receive update

\(<a0, b0, a1, b1>\)
\(<a0, a1, b0, b1>\)
\(<a0, a1, b1, b0>\)

What next?
Informal
Intuitive

Behaviour: occam-π (executable)

PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
           BARRIER bar)
  WHILE TRUE
    INT x, y, z:
      SEQ
        ask ? x       -- take question
        a0 ? y
        ans ! 0       -- return answer
        b0 ? z
        SYNC bar      -- wait for others
      c0 ! 0
  :

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
           BARRIER bar)
  WHILE TRUE
    INT x, y, z:
      SEQ
        ask ! 0       -- ask question
        ans ? x       -- wait for answer
        a1 ? y
        b1 ? z
        SYNC bar      -- wait for the others
      c1 ! 0
      ping ! 0      -- update neighbour
  :

PROC P2 (CHAN INT d0!, d1!, ping?,
           BARRIER bar)
  WHILE TRUE
    INT x:
      SEQ
        SYNC bar    -- wait for others
        d0 ! 0
        ping ? x    -- receive update
        SYNC bar    -- wait for others
        d1 ! 0
        ping ? x    -- receive update
  :

What next?

* any order

<a0, b0, a1, b1>
<a0, a1, b0, b1>
<a0, a1, b1, b0>
Informal
Intuitive

Behaviour: \texttt{occam-π} (executable)

\begin{verbatim}
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
         BARRIER bar)
  WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x     -- take question
    a0 ? y
    ans ! 0     -- return answer
    b0 ? z
    SYNC bar    -- wait for others
    c0 ! 0
  :

PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!,
         BARRIER bar)
  WHILE TRUE
  INT x, y, z:
  SEQ
    ask ! 0     -- ask question
    ans ? x     -- wait for answer
    a1 ? y
    b1 ? z
    SYNC bar    -- wait for others
    c1 ! 0
    ping ! 0    -- update neighbour
  :

PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
  INT x:
  SEQ
    SYNC bar    -- wait for others
    d0 ! 0
    ping ? x    -- receive update
    SYNC bar    -- wait for others
    d1 ! 0
    ping ? x    -- receive update
  :
\end{verbatim}

That’s \textbf{18} possible orderings of the first \textbf{7} signals.

What happens when the sub-processes start looping?
**Informal Intuitive**

**Behaviour: occam-Π (executable)**

```
PROCP1 (CHAN INT a1?, b1?, c1!, ask!, ans!,
    BARRIER bar)
WHILE TRUE
    INT x, y, z:
    SEQ
        ask ! 0     ---- ask question
        ans ? x     ---- wait for answer
        a1 ? y
        b1 ? z
        SYNC bar    ---- wait for others
        c1 ! 0
        ping ! 0    ---- update neighbour

PROCP0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
    BARRIER bar)
WHILE TRUE
    INT x, y, z:
    SEQ
        ask ? x     ---- take question
        a0 ? y
        ans ! 0     ---- return answer
        b0 ? z
        SYNC bar    ---- wait for others
        c0 ! 0

PROCP2 (CHAN INT d0!, d1!, ping?,
    BARRIER bar)
WHILE TRUE
    INT x:
    SEQ
        SYNC bar    ---- wait for others
        d0 ! 0
        ping ? x    ---- receive update
        SYNC bar    ---- wait for others
        d1 ! 0
        ping ? x    ---- receive update
```

Could P0 signal again on a0 before P2 gave its first d0?

Are there any more possible first-7 signal sequences?
With **verification qualifiers** and **assertions**, we can ask the *occam-\pi* compiler to **model check** the previous intuition (which was only about the opening behaviour of the system) and answer crucial questions **about its continuing behaviour**.

The programmer’s (or specifier/designer’s) skill will be to decide (and formulate) the right questions!

The compiler’s task is to generate $\text{CSP}_M$, a **declarative (functional)** language, from the *occam-\pi* source, use the (FDR2) model checker and return meaningful answers.
If we generated $\text{CSP}_M$ that fully reflected the semantics of the source code, we would quickly produce a system with too many states for any feasible \textit{model checking}. For instance, a single \texttt{INT} variable has 4G possible states!

\textbf{By default}, therefore, data values are ignored when generating the $\text{CSP}_M$. For instance:

\begin{verbatim}
PROC P (VAL INT i, CHAN INT c!) 
  c! i 
:
\end{verbatim}

maps just to:

\begin{verbatim}
P (c) = c -> SKIP
\end{verbatim}
Formal

Verify Qualifiers: data

\texttt{occam-\pi} code dependant on tests of \texttt{untracked} run-time values map to non-deterministic choice:

\begin{verbatim}
PROC Q (VAL INT i, CHAN INT c!, d!)
  IF
    i = 42
    c! i
  TRUE
    d! i
:
\end{verbatim}

maps to:

\begin{verbatim}
Q (c, d) = c -> SKIP |~| d -> SKIP
\end{verbatim}
If data values are significant, we qualify their types:

```plaintext
PROC Q (VAL VERIFY INT i, CHAN INT c!, d!)
  IF
    i = 42
    c ! i
    TRUE
    d ! i
  ::
```

Such data variables are tracked and the above now maps to:

```plaintext
Q (i, c, d) =
  if i == 42 then c -> SKIP else d -> SKIP
```
If data values are significant, we qualify their types:

```
PROC Q (VAL VERIFY INT i, CHAN VERIFY INT c!, d!)
  IF
    i = 42
    c ! i
    TRUE
    d ! i
  :;
```

Such data variables and channel messages are tracked and the above now maps to:

```
Q (i, c, d) =
  if i == 42 then c!i -> SKIP else d!i -> SKIP
```
Compiling: occam-π \( \rightarrow \) CSP\(_M\)

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!,
BARRIER bar)

WHILE TRUE
  INT x, y, z:
  SEQ
    ask ? x  -- take question
    a0 ? y
    ans ! 0
    b0 ? z
    SYNC bar  -- return answer
    c0 ! 0
    ...
```

\[ P0\,(a0, b0, c0, ask, ans, bar) = \]

\[
\text{let}
\]

\[
\text{let } P0_0_ = \text{ask } \rightarrow \text{a0 } \rightarrow \text{ans } \rightarrow \text{b0 } \rightarrow \text{bar } \rightarrow \text{c0 } \rightarrow P0_0_ \]

\[
\text{within}
\]

\[
P0_0_\]

Formal

Device

\begin{align*}
\text{a0} & \quad \text{b0} & \quad \text{c0} \\
\text{a1} & \quad \text{b1} & \quad \text{c1} \\
\text{d0} & \quad \text{d1} \\
\end{align*}
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, barbar)

WHILE TRUE

INT x, y, z:
SEQ
  ask ! 0
  ans ? x
  a1 ? y
  b1 ? z
  -- ask question
  SYNC bar
  -- wait for answer
  cl ! 0
  ping ! 0
  -- update neighbour

::

Pl (a1, b1, c1, ask, ans, ping, bar) =

let
  P1_0_ = ask -> ans -> a1 -> b1 -> bar -> c1 -> ping -> P1_0_
within
P1_0_
Compiling: \textit{occam-π} → \textit{CSP}_M

\begin{verbatim}
PROC P2 (CHAN INT d0!, d1!, ping?,
         BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar   -- wait for others
      d0 ! 0     -- receive update
      ping ? x   -- wait for others
      d1 ! 0     -- receive update
      ping ? x
    :
\end{verbatim}

\begin{verbatim}
P2 (d0, d1, ping, bar) =
  let
    \texttt{P2\_0\_} = bar -> d0 -> ping -> bar -> d1 -> ping -> \texttt{P2\_0\_}
  within
    \texttt{P2\_0\_}
\end{verbatim}
PROC Device (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
CHAN INT ask, ans, ping:
BARRIER bar:
PAR ENROLL bar
  P0 (a0?, b0?, c0!, ask?, ans!, bar)
  P1 (a1?, b1?, c1!, ask!, ans?, ping!, bar)
  P2 (d0!, d1!, ping?, bar)
Compiling: 

channel `ask_0`, `ans_0`, `ping_0`, `bar_0`

Device `(a0, b0, c0, a1, b1, c1, d0, d1)` =

```occam-pi
let Device_0 =
  ( P0 (a0, b0, c0, ask_0, ans_0, bar_0)
    || {ask_0, ans_0, bar_0} ||
    P1 (a1, b1, c1, ask_0, ans_0, ping_0, bar_0)
  )
\ {ask_0, ans_0}
within
  ( Device_0 || {ping_0, bar_0} ||
    P2 (d0, d1, ping_0, bar_0)
  )
\ {ping_0, bar_0}
```

local channels are declared globally, used locally, hidden and not used again
**Verify Assertions**: \( \texttt{occam-\pi} \)

Verify Assertions:

- `VERIFY <assertion>`
- `VERIFY NOT <assertion>`

### Assertions

- `DETERMINISTIC.F`<process>
- `DETERMINISTIC.FD`<process>
- `DEADLOCK.FREE.F`<process>
- `DEADLOCK.FREE.FD`<process>
- `LIVELOCK.FREE`<process>
- `TERMINATES`<process>

### Specials

- `REFINES.T`<process>
- `REFINES.F`<process>
- `REFINES.FD`<process>

---

Only `VAL VERIFY` operands need to be supplied (channels and barriers are supplied automatically).

where `<process>` is an instance of a `PROC`
Verify Assertions: \texttt{occam-π}

\begin{itemize}
  \item \texttt{VERIFY <assertion>}
  \item \texttt{VERIFY NOT <assertion>}
\end{itemize}

\begin{itemize}
  \item \texttt{DETERMINISTIC.F <process>}
  \item \texttt{DETERMINISTIC.FD <process>}
  \item \texttt{DEADLOCK.FREE.F <process>}
  \item \texttt{DEADLOCK.FREE.FD <process>}
  \item \texttt{LIVELOCK.FREE <process>}
  \item \texttt{TERMINATES <process>}
\end{itemize}

\begin{itemize}
  \item \texttt{EQUIVALENT.T <process>}
  \item \texttt{EQUIVALENT.F <process>}
  \item \texttt{EQUIVALENT.FD <process>}
\end{itemize}

Only \texttt{VAL VERIFY} operands need to be supplied (channels and barriers are supplied automatically)

where \texttt{<process>} is an instance of a \texttt{PROC}

Refinement in both directions!
Without testing the system, we can assert straight away that our **Device** is *deterministic* and *free from deadlock* and *livelock* – and that it doesn’t *terminate*:

```
VERIFY DETERMINISTIC.FD Device
VERIFY DEADLOCK.FREE.FD Device
VERIFY LIVELOCK.FREE Device
VERIFY NOT TERMINATES Device
```

and the compiler says: “✔”!
To verify behaviours beyond determinism, deadlock and livelock freedom and termination, we need some way to express the behaviours we want. We can use \texttt{occam-\pi} for this, together with \textit{refinement}.

\begin{verbatim}
VERIFY PROC P (...) 
  ... 
  :
\end{verbatim}

The \texttt{occam-\pi} compiler generates only \textit{CSP}_M from such declarations – no executable code.

Within \texttt{VERIFY} processes, certain restrictions \texttt{occam-\pi} imposes (currently) can be removed – for instance, \textit{output guards} and \textit{barrier guards} are allowed.

Only \texttt{VERIFY} processes can invoke \texttt{VERIFY} processes.
A *trace* of a process is a *finite* sequence of external events that it *can* perform.

Process $P$ *trace refines* $Q$ if all *traces* of $P$ are *traces* of $Q$.

<assertion>

$P \text{ REFINES }. T \ Q$

This is a *safety* result: it means that any trace $P$ performs is sanctioned by $Q$ (considered here as its specification). *It cannot perform unspecified patterns of behaviour.*
**Formal Behaviour: **\textit{occam-}\(\pi\) (*verifyable*)

To check whether particular event sequences (*traces*) may initially be performed by \textit{Device} ... e.g.

Define processes that have no choice in the matter ... e.g.

\begin{itemize}
\item \(<a_0, b_0, a_1, b_1>\)
\item \(<a_0, a_1, b_0, b_1>\)
\item \(<a_0, a_1, b_1, b_0>\)
\end{itemize}

\textbf{Informal understanding}

\textbf{What next?}

\begin{itemize}
\item \(c_0\)
\item \(c_1\)
\item \(d_0\)
\end{itemize}

(* any order)
**Formal Behaviour:** \texttt{occam-\pi} (verifyable)

```
VERIFY PROC T0 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  INT x:
  SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    b1 ? x
    d0 ! 0
    c0 ! 0
    c1 ! 0
  STOP
```

**Informal understanding**

Define processes that have no choice in the matter ... e.g.

- \(<a0, b0, a1, b1>\)
- \(<a0, a1, b0, b1>\)
- \(<a0, a1, b1, b0>\)

What next?

\(<a0, b0, a1, b1>\)
\(<a0, a1, b0, b1>\)
\(<a0, a1, b1, b0>\)

\((* \text{any order})\)

Now \(<a0, b0, a1, b1, d0, c0, c1>\) is, clearly, a trace of \texttt{T0}. Therefore, it is also a trace of \texttt{Device}.
**Formal**

**Behaviour: occam-π (verifyable)**

```
VERIFY PROC T0 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  INT x:
  SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    b1 ? x
    d0 ! 0
    c0 ! 0
    c1 ! 0
  STOP
```

**Informal understanding**

- `<a0, b0, a1, b1>`
- `<a0, a1, b0, b1>`
- `<a0, a1, b1, b0>`

What next?

- `c0`  
- `c1`  
- `d0`  

(* any order)

Define processes that have no choice in the matter ... e.g.

```
VERIFY T0 REFINES.T Device
```

... which verifies our intuition ☺☺☺
Formal Behaviour: **occam-π** (verifyable)

**Verify**

```occam-π```
```
VERIFY PROC T1 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
   INT x:
   SEQ
   a0 ? x
   b0 ? x
   a1 ? x
   d0 ! 0
   b1 ? x
   c0 ! 0
   c1 ! 0
   STOP
```

Informal understanding

Define processes that have no choice in the matter ... e.g.

```
VERfY T1 REFINES.T Device
```

What next?

- `c0`
- `c1`
- `d0`

(* any order)

A counter-example is provided:

```
<a0, b0, a1, d0>
```

is a trace of `T1`, but not `Device`.

these two lines were swapped
Formal

**Behaviour: occam-π (verifyable)**

```
VERIFY PROC T1 (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
    INT x:
    SEQ
    a0 ? x
    b0 ? x
    a1 ? x
    d0 ! 0
    b1 ? x
    c0 ! 0
    c1 ! 0
    STOP
```

Informal understanding

```
<?a0, b0, a1, b1>
<?a0, a1, b0, b1>
<?a0, a1, b1, b0>
```

What next?

- c0
- c1
- d0

(* any order)

Define processes that have no choice in the matter ... e.g.

```
VERIFY T1 REFINES.T Device
```

... which verifies our intuition 😊😊😊
Let's ask a more difficult question about the continuous running of the system. Suppose the robot would do something very bad if its controller Device were ever to perform a signal twice on a0 without a signal on d0 or d1 in between. Might this ever happen?

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming …
**Behaviour:** \textit{occam-π (verifyable)}

Simple: write a process that checks all signals to/from \textit{Device}, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming …
Formal Behaviour: occam-\pi \text{ (verifyable)}

Safety

Simple: write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming …
**Formal**

**Behaviour:** \texttt{occam-\pi} \texttt{(verifyable)}

\begin{verbatim}
VERIFY PROC Check (CHAN INT a0!, b0!, c0?, a1!, b1!, c1?,
      d0?, d1?, alive!)

  ---* n : the number of a0 signals since the last d0 or d1

INITIAL VERIFY INT n IS 0:
WHILE TRUE
  SEQ
    alive ! 0    IF
      n >= 2
        STOP -- refuse all further signals (forcing deadlock)
        TRUE
          ... process next signal (maintain n)

: 
\end{verbatim}

**Simple:** write a process that checks all signals to/from Device, looking for the bad scenario and deliberately deadlocking the monitored system if spotted. This is just programming ...
Formal

**Behaviour:** *occam-π* (verifyable)

{{{
process next signal (maintain n)

INT x:
ALT
  a0 ! 0
    n := n + 1
  b0 ! 0
    SKIP
  c0 ? x
    SKIP
  a1 ! 0
    SKIP
  b1 ! 0
    SKIP
  c1 ? x
    SKIP
  d0 ? x
    n := 0
  d1 ? x
    n := 0
}}}

$n = \text{the number of a0 signals received since the last } d0 \text{ or } d1$

This is an ALT with four input and four output guards

output guards are currently disallowed in *executable occam-π* (performance reasons)
**Formal Behaviour:** \( \text{occam-\pi} \) (verifyable)

**Safety**

```
VERIFY PROC CheckDevice (CHAN INT alive!)
  CHAN INT a0, a1, b0, b1, c0, c1, d0, d1:
  PAR
    Check (a0!, b0!, c0?, a1!, b1!, c1?, d0?, d1?, alive!)
    Device (a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
:```

```
CHAN INT a0, a1, b0, b1, c0, c1, d0, d1:
PAR
  Check (a0!, b0!, c0?, a1!, b1!, c1?, d0?, d1?, alive!)
  Device (a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
:
```
Formal Behaviour: \textit{occam-\pi} (verifyable)

Safety CheckDevice

---

If \textbf{Check} stops, \textbf{CheckDevice} will deadlock.
Therefore, \textbf{Check} never stops \textit{... and the bad thing can't happen.}

\textbf{Q.E.D.}
**Formal**

**Behaviour:** occam-$\pi$ (verifyable)

---

**Safety**

---

**Check**

---

**Device**

---

**Check Device**

---

**alive**

---

**VERIFY DEADLOCK.FREE.FD Check Device**

---

**Note:** protocol checking monitors, such as **Check**, are sometimes used live to ensure adherence at run-time (e.g. in device drivers). We are using **Check** purely for static analysis – it is not there at run-time and, therefore, has no impact on performance.
So far, our checks have concerned safety – namely that our system will not do harm (incorrect things). This is not enough! After all, the STOP process does not do incorrect things – it does nothing. STOP trace refines every process. Trace refinement is not enough.

A CSP failure is a state that a system reaches (represented by its trace to that point) and a set of events with which (if offered by its environment) it may refuse to synchronise.

Process $P$ failure refines $Q$ if (all traces of $P$ are traces of $Q$) and (all failures of $P$ are failures of $Q$).

$P \text{ REFINES.}F Q$
Failure refinement makes a powerful statement! \( P \) can only do traces of \( Q \) (so its safe). More: the failures of \( P \) are allowed by \( Q \). If \( P \) and \( Q \) execute the same trace to a state where their environment offers a set of events that \( Q \) will not refuse, then \( P \) also will not refuse.

A CSP failure is a state that a system reaches (represented by its trace to that point) and a set of events with which (if offered by its environment) it may refuse to synchronise.

Process \( P \) failure refines \( Q \) if (all traces of \( P \) are traces of \( Q \)) and (all failures of \( P \) are failures of \( Q \)).
We can describe “**P failure refines Q**” in a positive way: whenever Q stays alive (engaging with its environment), so does P (and in the same way). So, if Q is a specification explicitly defining the required patterns of synchronisation, P will provide them.

A **CSP failure** is a **state** that a system reaches (represented by its **trace** to that point) and a **set of events** with which (if offered by its environment) it **may refuse to synchronise**.

Process **P failure refines Q** if (all **traces** of P are **traces** of Q) and (all **failures** of P are **failures** of Q).
Behavour: \textit{occam-\pi} (verifyable)

Recall our informal understanding of (at least some of) the opening traces of Device \textit{(slides 18-35)} ...

We can formalise the expression of those traces a bit better ...

\begin{itemize}
  \item \langle a_0, b_0, a_1, b_1 \rangle
  \item \langle a_0, a_1, b_0, b_1 \rangle
  \item \langle a_0, a_1, b_1, b_0 \rangle
\end{itemize}
Recall our informal understanding of (at least some of) the opening traces of **Device** *(slides 18-35)* ... 

We can formalise the expression of those traces a bit better ...
Recall our informal understanding of (at least some of) the opening traces of **Device (slides 18-35) ...**

We can formalise the expression of those traces a bit better ...
Recall our informal understanding of (at least some of) the opening traces of **Device** *(slides 18-35)* …

We can formalise the expression of those traces a bit better …

\[
\langle a0 \rangle; (\langle b0 \rangle ||| \langle a1, b1 \rangle); (\langle c0 \rangle ||| \langle c1 \rangle ||| \langle d0 \rangle)
\]
And, *still using our intuitive understanding*,
guess the next cycle of events …

We can formalise the expression of
those traces a bit better …

\[
\langle a_0 \rangle; (\langle b_0 \rangle \parallel \langle a_1, b_1 \rangle); (\langle c_0 \rangle \parallel \langle c_1 \rangle \parallel \langle d_0 \rangle); \\
\langle a_0 \rangle; (\langle b_0 \rangle \parallel \langle a_1, b_1 \rangle); (\langle c_0 \rangle \parallel \langle c_1 \rangle \parallel \langle d_1 \rangle)
\]
And, still using our intuitive understanding, guess the next cycle of events …

We can formalise the expression of those traces a bit better …

And the rest …

\[
(a_0; (b_0 \mid\mid a_1, b_1); (c_0 \mid\mid c_1 \mid\mid d_0);) *
\]
From such trace expressions, we can directly write down an $\text{occam-\pi}$ process that offers all of them ...

\[
(<a_0>; (<b_0> ||| <a_1, b_1>); (<c_0> ||| <c_1> ||| <d_0>);) 
* 
(<a_0>; (<b_0> ||| <a_1, b_1>); (<c_0> ||| <c_1> ||| <d_1>))
\]
From such trace expressions, we can directly write down an *occam-π* process that offers all of them ...

\[
\left( \langle a_0 \rangle; \left( \langle b_0 \rangle ||| \langle a_1, b_1 \rangle \right) \right) \cdot \left( \langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_0 \rangle \right) \cdot \left( \langle a_0 \rangle; \left( \langle b_0 \rangle ||| \langle a_1, b_1 \rangle \right) \right) \cdot \left( \langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_1 \rangle \right)
\]
From such trace expressions, we can directly write down an \texttt{occam-π} process that offers all of them ...

\[
\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_0 \rangle);)\ast
\]

\[
\langle a_0 \rangle; (\langle b_0 \rangle ||| \langle a_1, b_1 \rangle); (\langle c_0 \rangle ||| \langle c_1 \rangle ||| \langle d_1 \rangle)
\]
Formal Behaviour: \texttt{occam-\pi} (verifyable)

\begin{verbatim}
{{{
  phase 1
  SEQ
    a0 ? w
  PAR
    b0 ? x
    SEQ
      a1 ? y
      b1 ? Z
    PAR

    c0 ! 0
    c1 ! 0
    d1 ! 0
}}}
\end{verbatim}

From such trace expressions, we can directly write down an \texttt{occam-\pi} process that offers all of them …

\begin{verbatim}
  (\langle a0\rangle; (\langle b0\rangle ||| \langle a1, b1\rangle); (\langle c0\rangle ||| \langle c1\rangle ||| \langle d0\rangle);) *
  (\langle a0\rangle; (\langle b0\rangle ||| \langle a1, b1\rangle); (\langle c0\rangle ||| \langle c1\rangle ||| \langle d1\rangle))
\end{verbatim}

This generation can be automated.
**Formal Behaviour: occam-\(\pi\) (verifyable)**

```
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, al?, b1?, c1!, d0!, d1!)
   ...
```

**DeviceSpec** is an explicit specification of all signal patterns we expect (or need) **Device** to be able to perform:

```
( <a0>; (<b0> || <a1, b1>); (<c0> || <c1> || <d0>); ) *
```

**Device** was not **implemented** as **DeviceSpec** because of the three independent functions (**weapons systems, vision processing** and **motion stability**) it had to perform. **Process-oriented design** led to its three communicating sub-systems.

Whilst our intuition indicated that the first two lines of **DeviceSpec** reflected the initial behaviour of **Device**, it was unclear whether the pattern repeated cleanly as its sub-components started looping.
**Formal Behaviour:** \textit{occam-π (verifyable)}

```plaintext
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
   ...
: 
```

**DeviceSpec** is an explicit specification of all signal patterns we expect (or need) **Device** to be able to perform:

\[
(<a0>; (<b0> ||| <a1, b1>); (<c0> ||| <c1> ||| <d0>);)\ast
\]

However:

```plaintext
VERIFY Device REFINES.F DeviceSpec
```

This is all we need. Any traces performed by **Device** are allowed by **DeviceSpec** – so it’s safe. Any failures reached by **Device** are allowed by **DeviceSpec** – so it’s as \textit{alive} as **DeviceSpec** (which was built always to offer everything in the specified trace pattern).
**Formal Behaviour:** **occam-π (verifyable)**

```plaintext
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  ...
: 
```

**DeviceSpec** is an explicit specification of all signal patterns we expect (or need) **Device** to be able to perform:

```plaintext
(<a0>; (<b0> ||<a1, b1>); (<c0> ||<c1> ||<d0>);)* 
(<a0>; (<b0> ||<a1, b1>); (<c0> ||<c1> ||<d1>))
```

However:

```plaintext
VERIFY Device REFINES.F DeviceSpec ✔
```

Without this verification, we may be tempted to add another barrier (bar) sync at the end of each loop of **P0** and **P1** and half-loop of **P2**. The above **refinement** shows that the required pattern does indeed repeat cleanly and, so, this overhead is unnecessary.
DeviceSpec is an explicit specification of all signal patterns we expect (or need) Device to be able to perform:

\[
\langle a_0 \rangle; \langle b_0 \rangle || \langle a_1, b_1 \rangle; \langle c_0 \rangle || \langle c_1 \rangle || \langle d_0 \rangle; \langle a_0 \rangle; \langle b_0 \rangle || \langle a_1, b_1 \rangle; \langle c_0 \rangle || \langle c_1 \rangle || \langle d_1 \rangle \] *
\]

However:

```
VERIFY Device REFINES.F DeviceSpec
```

Rather than being deduced after implementation, DeviceSpec may be part of the specification for Device. We certainly need assurance of the behaviour of Device to use it securely with other components. All its patterns of synchronisation (for safety and liveness questions) can be trivially deduced from DeviceSpec.
Formal Behaviour: **occam-π (verifyable)**

```plaintext
VERIFY PROC DeviceSpec (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
    ...
```

**DeviceSpec** is an explicit specification of all signal patterns we expect (or need) **Device** to be able to perform:

```
(<a0>; (<b0> ||| <a1, b1>); (<c0> ||| <c1> ||| <d0>);)*
(<a0>; (<b0> ||| <a1, b1>); (<c0> ||| <c1> ||| <d1>))
```

However:

```
VERIFY Device REFINES.F DeviceSpec  ✔
```

We also have:

```
VERIFY DeviceSpec REFINES.F Device  ✔
```

But that’s just icing on the cake! ☺ ☺ ☺
Formal

Verify Assertions: compilation

For simplicity, most process arguments are omitted in VERIFY assertions – the occam-π compiler supplies all necessary events:

```
VERIFY DEADLOCK.FREE.FD Device

channel a0_42_, b0_42_, c0_42_, a1_42_,
    b1_42_, c1_42_, d0_42_, d1_42_

assert Device (a0_42_, b0_42_, c0_42_, a1_42_,
    b1_42_, c1_42_, d0_42_, d1_42_):
    [ deadlock free [FD] ]
```

The CSP_m channel names are generated from the occam-π and occam-π parameter names of the asserted process, suffixed by a unique number generated by the compiler.
Formal

Verify Assertions: compilation

For simplicity, most process arguments are omitted in VERIFY assertions – the occam-π compiler supplies all necessary events:

```plaintext
VERIFY NOT TERMINATES Device

assert not SKIP [FD=
Device (a0_42_, b0_42_, c0_42_, a1_42_,
b1_42_, c1_42_, d0_42_, d1_42_) \ Events
```

The CSP\textsubscript{M} channel names are generated from the occam-π CHAN and BARRIER parameter names of the asserted process, suffixed by a unique number generated by the compiler.
For simplicity, most process **arguments** are omitted in **VERIFY** assertions – the **occam-π** compiler supplies all necessary **events**:

```
VERIFY NOT TERMINATES Device

assert not SKIP [FD=
 Device (a0_42, b0_42, c0_42, a1_42,
    b1_42, c1_42, d0_42, d1_42) \ Events
```

Subsequent assertions about the same process may reuse channels previously generated.
For simplicity, most process arguments are omitted in VERIFY assertions – the occam-π compiler supplies all necessary events:

```
VERIFY Device REFINES.FD DeviceSpec
```

```
assert DeviceSpec (a_0_42_, b_0_42_, c_0_42_, a_1_42_, b_1_42_, c_1_42_, d_0_42_, d_1_42_)

[FD=
 Device (a_0_42_, b_0_42_, c_0_42_, a_1_42_, b_1_42_, c_1_42_, d_0_42_, d_1_42_)
```

Subsequent assertions about the same process may reuse channels previously generated. [Note: processes in refinement assertions should have the same parameter signatures, though the formal names can be different].
The only arguments needed for CSP assertions are those for data parameters. Channels and barriers can be supplied automatically. Non-data parameters are irrelevant.

For example, if we need an assertion about:

```
PROC System (VAL INT n, CHAN INT out!)
```

we must supply a value for \(n\), since we have declared it relevant:

```
VERIFY DEADLOCK.FREE.FD System (42, _)
```

where the underscore indicates arguments that are either irrelevant (non-data) or automatic (channels and barriers).
Later, we plan an option for the occam-\(\pi\) compiler just to generate \(\text{CSP}_M\) code to be picked up by a GUI with facilities for interactive generation, checking and reporting of VERIFY assertions. These will be similar to those given by the FDR2 GUI, but processes and assertions will be in terms of the occam-\(\pi\) sources. FDR2, or some derivative, remains the underlying workhorse for model checking.

The GUI will allow flexible exploration of assertions with VERIFY data values. It will also prove useful when some assertions take a long time to check ... rather than wait for all checks to complete during compilation (as a single batch of assertions to FDR2).
Reflection on Case Study (Device)

Further study:

All sorts of *what-ifs* on the behaviour of the system can be explored and answered without running any code ... e.g.

If the (internal) **ping** communications were removed, does **Check** still hold?  
No

Do the **a0** and **a1** signals strictly alternate?  
Yes

Do the **b0** and **b1** signals strictly alternate?  
No

If we added an extra **bar** sync at the end of each cycle in **P0** and **P1** and half-cycle in **P2**, would it make any difference?  
No

If the elevator cabin is not at a floor, might the floor doors to the elevator shaft still open?  
Another exercise ...
Reflection

\texttt{occam-\pi} / \texttt{CSP}_M

\texttt{occam-\pi} teams well with \texttt{CSP}_M to provide efficient executables and rich formal analysis.

This presentation reflects a proposal to extend \texttt{occam-\pi} to include verification assertions (about deadlock, livelock, determinism and refinement). Its compiler will generate suitably abstracted \texttt{CSP}_M and interact with the FDR2 model checker, feeding back results in terms of the source \texttt{occam-\pi} program.

Together with the ancient formal Laws of \texttt{occam Programming}, this moves \texttt{occam-\pi} towards a process algebra in its own right.

* [A.W.Roscoe and C.A.R.Hoare, 1988]
Reflection

Observation

Formal verification of the behaviour of concurrent processes can be achieved – by students – engaging in only simple reasoning themselves.

The complexity of synchronisation and communication analysed goes far beyond the embarrassingly parallel.

Aside: model checking found an error overlooked in developing the (Device) case study on paper (the need for ping) … which shows the necessity for formal checks (especially when those responsible think they won’t make mistakes!).

Further reading: Santa Claus: Formal Analysis of a Process Oriented Solution *

* http://doi.acm.org/10.1145/1734206.1734211

TOPLAS, [April, 2010]
Reflection

Class experience

The (Device) case study presented was developed from one first worked through in a single lesson of a graduate class in concurrency at UNLV in the spring of 2010.

They had previously studied a range of concurrency approaches, including process-oriented material from the Kent “Concurrency Design and Practice” course.

They were comfortable with using occam-π in non-trivial projects (thousands of interacting processes), so the example system here would be considered fairly simple.

Nevertheless, it was appreciated that relying just on intuitive understanding is unsafe – especially if the application were safety critical.
A Thesis (for which we have experimental evidence)

Not only can we (and should we) teach concurrency at the start of the undergraduate CS curriculum …

But also can (and we should) teach formal analysis and verification of this concurrency at the same time … and it can be done just by ordinary programming (execution code + verification code). ☺
Not only can we (and should we) teach concurrency at the start of the undergraduate CS curriculum …

Because it’s there:

**Fundamental primitives** for software engineering

*All* are important. *All* are simple. *All* are available.
**A Thesis** *(for which we have experimental evidence)*

Not only *can* we (and *should* we) teach concurrency at the **start** of the undergraduate CS curriculum …

- **Because it’s there**
- **Because it simplifies**
- **Because it scales**

**CSP / π-calculus**

**occam-π / JCSP**

- **for complexity**
- **for performance**
A Thesis (for which we have experimental evidence)

Complex and high-performance systems cannot avoid concurrent design, implementation and reasoning.

Common concurrency bugs are intermittent – not repeatable on demand. Untestable in practice.

We stand on the shoulders of giants (who made the theory and model checkers). We verify programs just by writing programs ... it becomes everyday practice.

But also we can (and we should) teach formal analysis and verification of this concurrency at the same time ... and it can be done just by ordinary programming (execution code + verification code).
Observation

Can we teach students *(those who love to program, anyway)* concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. processes, communication, synchronisation, networks) and higher level patterns (e.g. client-server, phased barrier, I/O-PAR) … ?

- they can use those primitives and patterns with the same fluency as they use serial computing primitives, *without tripping over dark hazards* … ?

- they can develop their own patterns when the standard ones don’t apply … ?

- they can use formal methods to verify good behaviour (*e.g. freedom from deadlock and livelock, safety, liveness*), without training in the underlying mathematics (*process algebra, denotational semantics*) … ?

- they can do this as *normal everyday practice*, without any sense of fear … ?
Observation

Can we teach students *(those who love to program, anyway)* concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. *processes*, *communication*, *synchronisation*, *networks*) and higher level patterns (e.g. *client-server*, *phased barrier*, *I/O-PAR*) …?

- they can use those primitives and patterns with the same fluency as they use serial computing primitives, *without tripping over dark hazards* …?

- they can develop their own patterns when the standard ones don’t apply …?

- they can use formal methods to verify good behaviour (e.g. *freedom from deadlock and livelock*, *safety*, *liveness*), without training in the underlying mathematics (*process algebra*, *denotational semantics*) …?

- they can do this *as normal everyday practice*, without any sense of fear …?
Can we teach students (those who love to program, anyway) concurrency so that:

- they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. processes, communication, synchronisation, networks) and higher level patterns (e.g. client-server, phased barrier, I/O-PAR) …?
- they can use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over dark hazards …?
- they can develop their own patterns when the standard ones don’t apply …?
- they can use formal methods to verify good behaviour (e.g. freedom from deadlock and livelock, safety, liveness), without training in the underlying mathematics (process algebra, denotational semantics) …?
- they can do this as normal everyday practice, without any sense of fear …?

Yes, we can!
Postscript ...

occam-π has ...

- a *dynamic concurrency model* built into its core design … with full denotational semantics (based on the *CSP traces/failures/divergences model*) …
- no *data race hazards* (eliminated by compiler aliasing analysis) …
- *deterministic concurrency* by default. Non-determinism is introduced *only* by explicit use of special features (e.g. choice, shared channels) …
- the *fastest and most effective* multicore scheduler on the planet (*maybe*) …
- *program verification by programming* (and a little thinking) …
- *simple to learn, simple to use* (e.g. 90 min Lego Robots occam workshop) …
- a tiny user base … to be fixed (???) … how (???) … when (???) …
Postscript ...

**occam-π** has ...

a dynamic concurrency model built into its core design ... with full denotational semantics (based on the CSP traces/failures/divergences model) ...

no data race hazards (eliminated by compiler aliasing analysis) ...

deterministic concurrency only by explicit use of special features (e.g. choice, shared channels) ...

the fastest and most effective multicore scheduler on the planet (maybe) ...

program verification by programming ... (and a little thinking) ...

simple to learn, simple to use (e.g. 90 min Lego Robots occam workshop) ...

But occam-π needs rationalising ...

It's time, again, for Occam's razor ...

a tiny user base ... to be fixed (?) ... how (???) ... when (???) ...